

Spaceport Command and Control System - Support
Software Development

Shayne Tremblay

National Aeronautics and Space Administration Kennedy Space Center

Computer Science

NIFS Spring Session

04/08/2016

Spaceport Command and Control System - Support Software Development

Shayne Tremblay

University of Central Florida, Orlando, Florida, 32816

The Information Architecture Support (IAS) Team, the component of the Spaceport Command and Control System (SCCS) that is in charge of all the pre-runtime data, was in need of some report features to be added to their internal web application, Information Architecture (IA). Development of these reports is crucial for the speed and productivity of the development team, as they are needed to quickly and efficiently make specific and complicated data requests against the massive IA database. These reports were being put on the back burner, as other development of IA was prioritized over them, but the need for them resulted in internships being created to fill this need. The creation of these reports required learning Ruby on Rails development, along with related web technologies, and they will continue to serve IAS and other support software teams and their IA data needs.

Nomenclature

CSCI	=	Computer Software Configuration Item
CUI	=	Compact Unique Identifier
DA	=	Development Activity
EIC	=	End Item Controller
ERB	=	Embedded Ruby
GUI	=	Graphical User Interface
HTML	=	HyperText Markup Language
IA	=	Information Architecture
IAS	=	Information Architecture Support
IDE	=	Integrated Development Environment
MPCV	=	Multi-Purpose Crew Vehicle
MVC	=	Model View Controller
SCCS	=	Spaceport Command and Control System
SLS	=	Space Launch System
TCID	=	Test Configuration Identifier

NASA NIFS – Internship Final Report

I. Introduction to Information Architecture

Information Architecture is a subteam of SCCS that is in charge of an internal web application, also called Information Architecture (IA). The IA application is essentially a web interface that allows users to view, upload, download, and generally interact with data related to components, CUIs, TCIDs, and various other types of data involved with NASA projects. Typical users include the various CSCI teams, engineers, and the IA team itself. To outsiders, IA can be considered a "magic black box" that can respond to a wide variety of very specific data requests, crucial for the development of both hardware and software projects across SCCS.

II. Objective

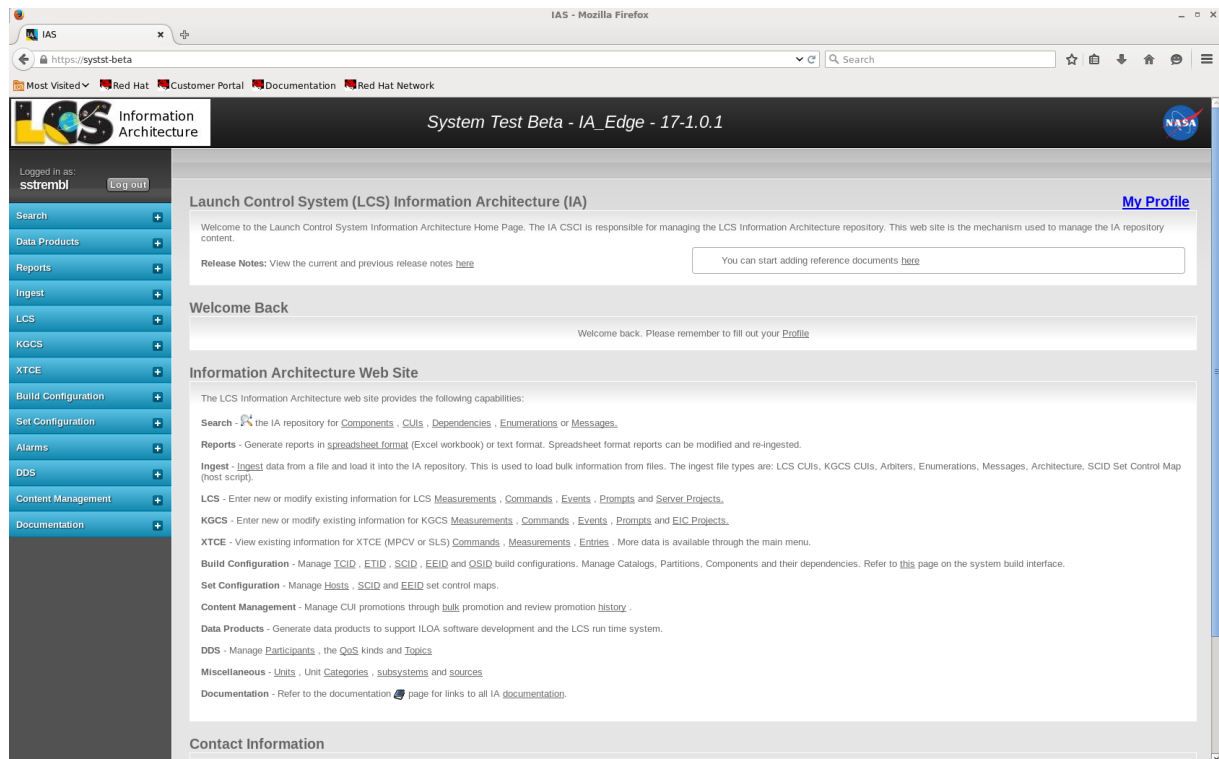


Figure 1. The Information Architecture (IA) application web interface. (Author owned image)

The internship position that I was chosen for was created in order to aid in the development of the Reports section of IA. IA reports are generally used by the SCCS software development teams to gather data for their projects about different components, CUIs, and various other pieces of data that are involved with SCCS projects. An example use case of the reports section could be: an engineer on a software team wants to see what components use a specific CUI, so they run a report against that CUI and get back a list of the components that use that CUI. These reports greatly improve productivity and efficiency in many software development projects, as they spare users from having to manually and systematically check every component and CUI that they are concerned with. My position involved the creation of these reports, the alteration of already existing reports, and the testing of these reports and related parts of the web application.

III. Technical Approach

A. Framework: Ruby on Rails

The IA web application is written using the Ruby on Rails web framework, an open-source web development framework built upon the Ruby programming language. Ruby is a high-level, object-oriented language that is known for its simplicity and readability. The Ruby community has created a multitude of reusable Ruby packages, called gems, which are free to use in Ruby applications. Rails is one of these gems, and it sets up a structured foundation for developing web applications quickly and easily via its “Convention over Configuration” methodology. Rails uses the popular model, view, controller (MVC) organizational pattern that is common among many web frameworks. Simply put, the model component contains backend “business” logic, the views contain the markup that the user will be interfacing with, and the controllers assist in the interaction between the models and views. Ruby on Rails thrives as a framework when it comes to data driven applications, which IA is the epitome of, due to its use of Active Record, an abstracted, object relational interface that allows developers to interact with a database in an easy, intuitive, and robust way. Active record accomplishes this via the way it “connects classes to relational database tables to establish an almost zero-configuration persistence layer for applications”.

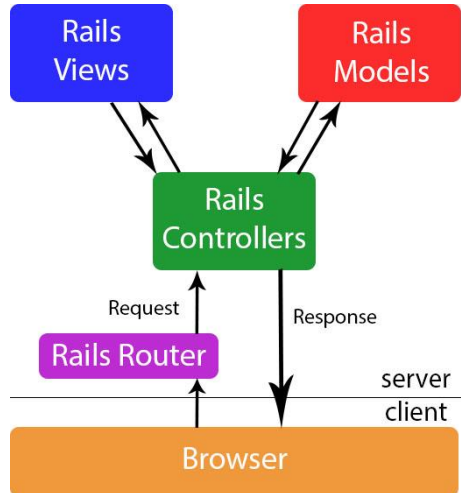


Figure 2. Ruby on Rails framework structure (author owned image)

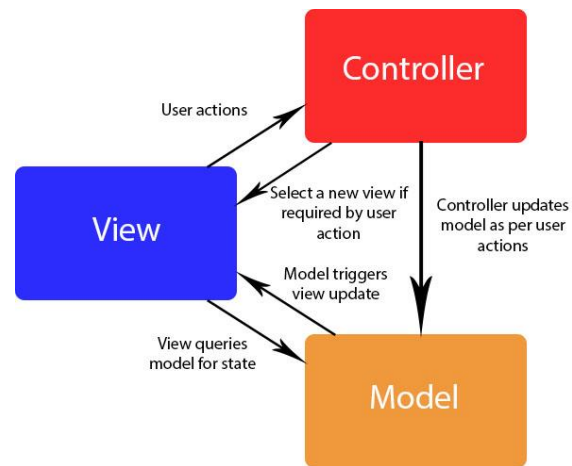


Figure 3. Model View Controller (MVC) structure (author owned image))

B. Frontend: CoffeeScript and Haml

Though Ruby on Rails is a full stack web framework, it falls short when it comes to dynamic front-end interfaces. This is where CoffeeScript and Haml come in. CoffeeScript is a programming language that feels extremely similar to Ruby, but compiles to JavaScript, and is commonly used for the front-end development of Rails applications. CoffeeScript files are kept in their own directory within the Rails application, and are used for situations such as having some content disappear when a checkbox is checked, etc. Haml is a markup language specifically used in Rails applications that provides a cleaner, more concise syntax, and replaces the Rails standard ERB frontend markup. ERB allows for the execution of Ruby code directly in markup code, and Haml allows for this too, but in a simpler, less cumbersome implementation.

NASA NIFS – Internship Final Report

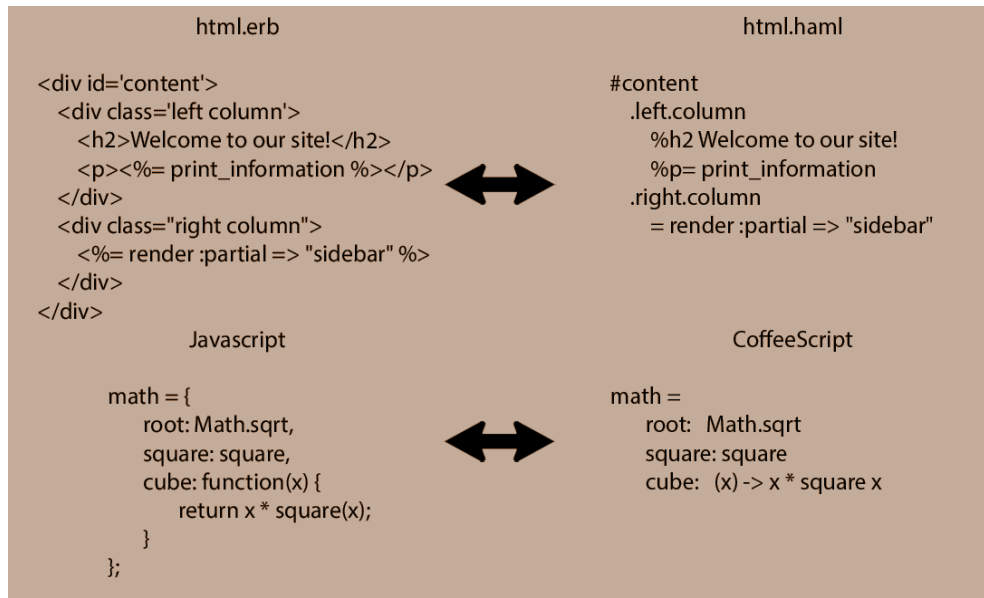


Figure 2. Comparison of HTML and Haml, and JavaScript and CoffeeScript. (author owned image)

C. Testing: RSpec and Cucumber

For unit testing, IA uses RSpec, a “behavior driven” testing framework for Ruby applications. The RSpec testing implementation is based on establishing context for what you are testing, and then assertions for the actual tests. For interface testing, IA uses Cucumber, another “behavior driven” testing framework. Cucumber tests are written in two parts: the first part is the feature file, which is written in seemingly plain English and provides a list of instructions for the test procedure, and the second part, which is the actually Ruby/RSpec code that the plain English instructions map to when the tests are actually run. Using these two frameworks, IA’s test suite covers everything from the user interface, to the data manipulation at the code level.

```
Feature:
  Users should be able to log in

Background:
  Given my browser is full screen

Scenario: Log in
  Given I have an account
  When I try to visit the root page
  Then I am asked to log in
  When I submit my log in info
  Then I should be on the root page

When(/^I submit my log in info$/) do
  fill_in 'Email', with: @email
  fill_in 'Password', with: @password
  find('input[value="Log in"]').click
end

Then(/^I should be on the root page$/) do
  expect(page).to have_content('Tasks')
  expect(page).to have_content('Completed Tasks')
end
```

Figure 3. Example of Cucumber feature and step code (author owned image)

D. IDE and Debugging: RubyMine and Byebug

RubyMine is a Ruby specific IDE that contains a lot of helpful features for Ruby and Rails application development. One of the most useful functions is the ability to click on a Ruby method call and get redirected to the declaration of that method. RubyMine also aids in refactoring; say you need to rename a variable or method, RubyMine can search your entire project to find and replace all calls/declarations of said variable or method. A tool that has proved invaluable in this position has been the Byebug gem, which is a Ruby application debugger. Byebug allows you to place breakpoints in your application code, which will freeze the execution at that point, and allow you to inspect variable values, change values, and even call methods in that frozen state. This makes investigating bugs and generally understanding application processes more approachable and intuitive.



```
[5, 14] in /Users/shayne/ruby/misc/byebug_output.rb
5:   if bar
6:     array << (1..1000).map{|i| "bar!"}
7:   else
8:     puts "no bar"
9:   end
10:  byebug
=> 11:  array
12: end
13:
14:  fooFunction('false')
(byebug)
```

Figure 4. Example of terminal output of Byebug gem. (author owned image)

E. Version Control and Code Reviews: AccuRev and Code Collaborator

IA is a massive application with an equally massive codebase, making version control and code reviews a must. The IA teams uses AccuRev for its version control, which is a stream-based version control system that has a GUI interface. AccuRev is also utilized by IA to supply data for safety-critical systems. Code is kept in AccuRev stations, and developers work off these stations (repositories of code) on their local machines via a workspace. Stations are organized hierarchically via streams; stations that are downstream from another station will pick up all of the changes of the upstream station when they update their workspace. This system allows for a controlled push of code to the very top stream, which is the stream from which the application itself is built. When code is promoted, it must go through the code review processes prior to being pushed upstream. This is done via Code Collaborator, a code review web application that can import AccuRev transactions and allow other users to leave comments, report bugs, and ultimately accept the code to be pushed upstream.

IV. Reports

A. Dependency Checking for TCID Report

The very first DA assigned to me was to fulfill a need for the already existing Dependency Checking for TCID Report. This report takes a TCID Definition as an input, and returns a file that contains a list of all the TCID Definition's dependencies, where a dependency is a link between a component and a CUI. In the old report, each row represents a dependency; the rows would contain the component name, the CUI name, and whether or not the CUI was included in the TCID Definition. If the CUI was included, the row would also contain the CUI's creation date, source, maturity, work order, author, and whether or not it is retired. If the CUI was not included in the TCID Definition, these columns were ignored for that dependency.

The DA gave a list of new requirements for this report, which I was to implement. The first being the addition of a column that states whether or not the component was verified, as well as the pedigree of the component. Adding these was simply a matter of altering the report generating method to pull this data from the already accessed component objects and printing them to the report. The next requirement was for all of the fields to be filled, regardless of whether or not the CUI was included in the TCID Definition. The solution to this was less trivial, as a rewrite of the majority of the previous implementation was required in order to have access to the database records containing the data for these missing CUIs. The method I used was to first deal with the CUIs that were included in the TCID Definition, and while pulling their data, I pushed the IDs of the missing CUIs to an array, which I used to query for them later on. The final report-generating class contained a lot of database queries, and with this came some performance concerns. In many cases, there was a need to query for a certain type of record, like a component record, but then pull data from an association to that record. This can result in $O(n^2)$ efficiency which means that the amount of time it would take for the query to run would grow exponentially in relation to the amount of data it had to query.

Active Record solves this problem by allowing for eager loading via a call to the 'includes' method, which loads the association into memory during the initial database query, allowing for quicker access to it later that does not involve any more database queries. The final product was not only an updated report that contained all of the data necessary to meet the requirements of the DA, but the report also runs faster than the previous version due to a more efficient setup of queries and data fetching loops.

```
# Fetches the CUIs (info_items) for all the missing CUIs and stores them as a hash where the
# CUI id (info_item identifier_id) is used as the key
missing_cuis = InfoItem.includes(:source, :identifier)
                    .where(identifier_id: not_included_ids).group_by(&:identifier_id)
```

Figure 7. Example of Active Record Query with use of 'includes' for eager loading. (author owned image)

Another requirement for the updated report was for users of the report to have the option to select either a full or violations report, where a full report would contain all of the TCID Definition's dependencies, and a violations report would only contain the dependencies where the component is not verified. It was also requested that, in the event the user chose to generate a violations report, the user can filter the types of violated dependencies that they want to show up in the report. The violations options include:

- "Failed CUI dependencies": This violation occurs when the CUI is missing in the TCID Definition
- "Reverification Needed": This is a cover for all component violations.
- "Retired CUIs": This violation occurs when a CUI is retired.

NASA NIFS – Internship Final Report

Implementing this requirement involved changing both the business logic and the user interface. The controller and model classes for this report were updated to be able to handle the extra parameters for the users' options, which was done via a few conditional statements. The user interface was previously a single dropdown menu, but the updated report now contains a second drop down menu with options for a "Full Report" and a "Violations Report". When the "Violations Report" option is selected, three check boxes appear, each representing one of the three violation types listed above, which was easy to implement via the use of CoffeeScript.

The figure displays two versions of a web form for 'Dependency Checking for TCID'. Both forms feature a 'TCID Definition' dropdown menu at the top. Below this is a 'Full or Violations Report' dropdown menu. In the left version, this menu is set to 'Full Report'. In the right version, it is set to 'Violations Report'. When 'Violations Report' is selected, three checkboxes appear: 'Failed CUI dependencies' (checked), 'Reverification Needed' (checked), and 'Retired CUIs' (unchecked). Both forms conclude with a blue 'Generate' button.

Figure 8. Dependency Checking for TCID user interface (author owned image)

Upon finishing up the updates to the report, another requirement outside of the original DA emerged that called for the report to be included in the TCID Build procedure. During the TCID Build procedure, the report would be run twice, once for a full report and again for a violations report, and the two resulting output files would be placed into a log directory. This required the creation of another method which would handle the call by the procedure and to set up the necessary parameters the user would normally have access to if running the report the typical way. The clients for this implementation also requested that the output files be in a .txt form, as the report would initially deliver a .csv file instead. Since .txt files are not tab delimited the way .csv files are, making this change involving utilizing one of Ruby's string formatting methods called 'ljust', which I used to dynamically append spaces to the end of strings to keep spacing standard for each column of the report.

The IA development team aims for extensive code coverage of the entire IA app, so when a feature is added, tests must also be added, and when a feature is altered, the tests for that feature must also be altered. Since IA features are very dependent on complicated data sets, and it is crucial for the accuracy of its different features, the FactoryGirl Ruby gem is used to generate sample data in order to validate accurate results. For the Dependency Checking for TCID Report, FactoryGirl is used to create a TCID Definition, various components, CUIs, and dependencies between those components and CUIs. The gem makes this easy by allowing for option parameters to be passed to the creation method in order to dynamically create data with specific requirements. For the unit tests, I used FactoryGirl to create dependencies that would match each use case and validated the resulting .txt file against the expected result. For the user interface tests, I used Cucumber's JavaScript flag, which causes it to run the tests in actual browser to validate that the user would be able to use the dropdowns and checkboxes as intended.

The finished Dependency Checking for TCID Report was a great success. It met all of the new requirements in terms of the data that is collected, the user interface, and its injection into the TCID Build process. A frequent user of this report is the SCCS Ops Build Team, who I met with frequently to understand their needs for the report's update. The initial report was helpful for them, but the missing data caused them to still have to manually check for some bits of crucial information, which was very time consuming. They were very satisfied with the updated report

NASA NIFS – Internship Final Report

and claimed that it would save them a considerable amount of time on a regular basis. The report will be available to the Ops Build Team, as well as all other IA users, in a newer iteration of IA coming in the near future.

B. Component Interdependency Report

My second assignment was to create an entirely new report, which was to be called the Component Interdependency Report. The goal for this report was to have a fast and simple way of seeing which components have dependencies on the same CUIs as a specific given component. In the event that someone was planning on removing a component from a project, it would be convenient for them to be able to see what components would be affected by the removal of the removed component's CUIs. The SCCS Ops Build Team needs to do this very thing on a regular basis, and actually used an informal script to retrieve this data when they needed. The script was lacking however, as it only provided them with component names. It was requested that the final report list not only the component name, but also the component type, the component's pedigree (i.e. preproduction, production, etc.), and the specific component with which it has the interdependency..

The development of this report began with creating the user interface. The DA stated that the input for a report will be a component of the type EIC Project, Server Project, MPCV, or SLS, and that the user can input one or more of these components. The implementation I decided to go with included two drop downs, where the first would have options for EIC Project, Server Project, MPCV, and SLS, and the second would contain all of the components for that first selection. When a user selects a specific component in the second drop down, they can click the “Add” button to add it to the “Selected Components” list, and the components in this list will be what the report runs against when the user clicks “Generate”. Adding the components to the list, and having this list get passed to the controller method required some pretty complicated CoffeeScript to get working. When the “Add” button is clicked, it not only appends the component name to the list, but also adds the component's ID to a hidden field which gets passed to the controller method.

Once the controller action is invoked, the IDs are stored in a hash where they are sorted by the types of components they are, and another hash is generated to act as a look up table for the original component names. This is necessary in order to maintain the link between the components found to have interdependencies and the component that they have an interdependency with. Once these hashes are created, they are passed to the report's model where the actual report is generated and where the database is queried. I based the querying procedure off of the informal query the Ops Build Team used when they needed this data, but I modified it to also include the newly requested columns. A few separate loops were also needed to organize the data in such a way for it to be printed to the resulting .txt file in the right format and order.

The most challenging part of creating this report was understanding the two different ways these dependencies existed in the IA database. The majority of the dependencies between components and CUIs was via a simple Dependency record which was associated with the given component and CUI. However, components and CUIs can also be associated via an IO Tag Name, which requires reaching through the database to the component's source to actually find. The final report will take each inputted component, find all of its dependent CUIs' IDs first, find all of the CUIs' IDs that have dependencies via IO Tag Names, and then proceed to find all of the components that have dependencies on this collection of CUIs. Once these components have been collected, they are sorted alphabetically, and the necessary data is pulled from these components and printed to a .txt file, which is the final product of the report.

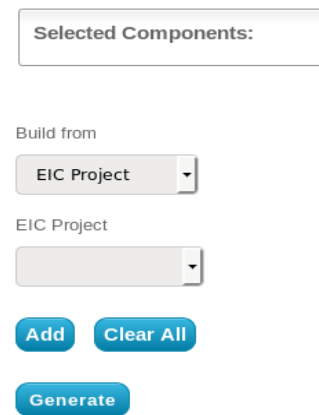


Figure 9. Component Interdependency report user interface (author owned image)

NASA NIFS – Internship Final Report

Testing for this report was very similar to the testing for the Dependency Checking for TCID Report. FactoryGirl was used to generate a handful of components, CUIs, and dependencies between these components and CUIs, and the unit test asserted that the proper data showed up in the final .txt file. The unit tests also check that an error occurs if the path that the model tries to save .txt to is invalid. RSpec makes this easy, where you can encapsulate code you expect to fail in a curly braces, and tell it that you expect an error from this code. The user interface testing for this report was a bit more robust. CoffeeScript can prove to be quite fragile and break easily, so the test involves testing the variety of input cases a user may have for this report.

Unlike the Dependency Checking for TCID Report, the Component Interdependency Report had one target customer, which was the SCCS Ops Build Team, who I met with frequently for this report as well. Because of this, I was able to complete this report far more quickly and efficiently, as there were fewer last minute changes to be made, which not only take time to implement, but also take more time to be code reviewed. The Ops Build Team was very satisfied with the result and are excited to use the report, as it will save the a lot of time in the same way the Dependency Checking for TCID Report will.

V. Conclusions

During this internship, I learned about purely data-driven uses for the Ruby on Rails framework and its related technologies. All of my previous Ruby on Rails experience was within the scope of being application based or for social networking. Compared to other frameworks, Rails is robust and scalable to where increasing the amount of and complexity of data is no issue, and the simplicity of Rails development makes it easy and efficient to add features. I believe that I would not have been able to begin contributing to the IA web application as fast as I did if it was not for how intuitive and approachable Ruby on Rails development is. I had very little database experience prior to starting this internship, and through the very data-oriented DAs that I worked, I am now very comfortable with database work, specifically the Active Record database interface. Collaboration is a huge component of IA development, where communication with the rest of the team, especially the more senior developers, is key to getting any work done. My communications skills have been increased as equally to, if not more so than, my web development skills.

Acknowledgments

I would like to give thanks to those that I had the pleasure to work with and receive amazing support from. This internship has been my first professional position, and it was nothing less than an honor for it to be at the Kennedy Space Center. Those who made it especially great include Andrew Davis, Logan Smith, Peata Ameperosa, Jared Rodriguez-Rivera, Josh Johnson, George Meyer, Oscar Brooks, Caylyne Shelton and Jamie Szafran.

References

Software:

Yukihiro "Matz" Matsumoto, "Ruby", <https://www.ruby-lang.org>

Rails Core Team, "Ruby on Rails", <http://rubyonrails.org/>

Jeremy Ashkenas, "CoffeeScript", <http://coffeescript.org/>

Norman Clarke, Matt Wildig, Akira Matsuda, Tee Parham, Nick Walsh, "Haml", <http://haml.info/>

David Chelimsky, Myron Marston, Andy Lindeman, Jon Rowe, Paul Casaretto, Sam Phippen, Bradley Schaefer, "RSpec", <http://rspec.info/>

Aslak Hellesøy, Joseph Wilk, Matt Wynne, Gregory Hnatiuk, Mike Sassak, "Cucumber", <https://cucumber.io/>

jetBrains, "RubyMine", <https://www.jetbrains.com/ruby/>

Deivid Rodriguez, "Byebug", <https://github.com/deivid-rodriguez/byebug>

AccuRev, Inc., "AccuRev", <http://www.borland.com/en-GB/Products/Change-Management/AccuRev>

SmartBear, "CodeCollaborator",
https://smartbear.com/product/collaborator/overview/#_ga=1.265703819.525207949.1460390061